Higher-Order Patterns in Replicated Data Types

Adriaan Leijnse
adriaan.leijnse@gmail.com
INESC TEC

Paulo Sérgio Almeida
psa@di.uminho.pt
INESC TEC & University of Minho

Carlos Baquero
cbm@di.uminho.pt
INESC TEC & University of Minho

ABSTRACT

The design of Conflict-free Replicated Data Types traditionally requires implementing new designs from scratch to meet a desired behavior. Although there are composition rules that can guide the process, there has not been a lot of work explaining how existing data types relate to each other, nor work that factors out common patterns. To bring clarity to the field we explain underlying patterns that are common to flags, sets, and registers. The identified patterns are succinct and composable, which gives them the power to explain both current designs and open up the space for new ones.

ACM Reference Format:
Adriaan Leijnse, Paulo Sérgio Almeida, and Carlos Baquero. 2019. Higher-Order Patterns in Replicated Data Types. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Conflict-free replicated data types (CRDTs) are concurrent variants of sequential data types that specify the outcome of conflicting concurrent operations. For operations on sets, priority can be given to either adds or removes, and for flags the same needs to happen for concurrent enabling and disabling of the flag. Consider the following definitions (notation is detailed in Section 2) for the add-wins set (1) and enable-wins flag (2) CRDTs:

$$\textsc{AWSet}(E) = \{\, a \mid e \in E \wedge e.\mathsf{o} = \langle \mathsf{add}, a\rangle \wedge \nexists e' \in E \cdot e'.\mathsf{o} = \langle \mathsf{rmv}, a\rangle \wedge e \prec e' \,\} \quad (1)$$

$$\textsc{EWFlag}(E) = \exists e \in E \cdot e.\mathsf{o} = \mathtt{enable} \wedge \nexists e' \in E \cdot e'.\mathsf{o} = \mathtt{disable} \wedge e \prec e' \quad (2)$$

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Why do these two definitions look so similar? Could we define one in terms of the other?

In this paper we want to argue that yes, we can, and that replicated data types in fact contain many examples of simple data types which can be used to define more complex ones. We hope that bringing these patterns to light will help solve an inefficiency of the current state of the art, namely that we tend to define and implement each CRDT from scratch despite many CRDTs sharing such structural patterns [1–4].

In order to illustrate our approach we first provide a brief intuition for CRDTs in Section 2. In the same section we also introduce the formalism used above, which gives each CRDT a simple denotation in the form of a function from a set of events registered by a replica in a distributed system to the current value of a CRDT for that replica.

Using this formalism we then illustrate how we can abstract over patterns in traditional definitions of CRDTs by carefully restructuring the latter. Reworking the definitions gives us *higher-order denotations*, that is, denotations which are functions from denotation to denotation. Some of our further contributions are:

- In Section 3.1 we use simple flags as a basis to derive the pattern in which CRDTs only consider the concurrent events to derive a value. These patterns are common to *causal* CRDTs.
- Last-writer-wins CRDTs discussed in Section 3.2 share a similar pattern with causal CRDTs, but resort to wall-clock timestamps rather than causality information.
- We introduce two new primitive flags, the enable-once and disable-once flags, which can be used to derive the more common enable-wins and disable-wins flags, as well as last-writer-wins variants of these flags.
- Section 4 defines set CRDTs in terms of Boolean flag CRDTs.
- Finally, in Section 5 we define a number of simple CRDTs which we use to obtain both concise and precise denotations for a large number of CRDTs by using the previously mentioned higher-order denotations, thereby demonstrating their power.

2 BACKGROUND

A CRDT [5] is a data type for replicated mutable state in a distributed system. Processes in the distributed system attempt to mutate the state by performing operations on the data

locally and transmitting the operations to other processes when possible. The value of a CRDT instance can differ for every process and depends on the operations received by that process.[1] However, two processes which have received the same operations will have the same value for the CRDT instance: a property called *eventual consistency*.

Since the value of a CRDT depends on the received operations, we formalize the behavior of a particular CRDT as a function $E \rightarrow V$, which we will call its *denotation*. The set E is the set of *events*: values which uniquely identify operations in time and space, while V is the set of values the CRDT can assume. We assume every event $e \in E$ has some properties which will be useful for the definition of various types of CRDTs later:

- $e.\mathtt{i}$ is a unique identifier for the event ($e.\mathtt{i}$ always uniquely identifies e, for any possible set E),
- $e.\mathtt{o}$ is the operation performed,
- $e.\mathtt{t}$ is the local wall-clock time of the replica when it originated the event,
- $e.\mathsf{k}$ is the set of events which were known at the origin replica when the event occurred, either directly or transitively (a process receiving an event e now also "knows" $e.\mathsf{k}$).

To allow us to be concise in our definitions we introduce the following notation for comparing properties of two uniquely identified events e and e':

- Timestamp comparison: $e < e' = e.\mathtt{t} < e'.\mathtt{t}$.
- Causality relation: $e \prec e' = e.\mathsf{k} \subset e'.\mathsf{k}$. The causality relation holds when one event e might have caused another due to it being known by the process producing e'. If this is the case we say that e is in the *causal past* of e'. If neither $e \prec e'$ nor $e' \prec e$ then we call e and e' *concurrent* events.

3 LWW AND CAUSAL FLAGS

Boolean flags are arguably among the simplest replicated data types. They have but two operations: `enable` and `disable`, and as their name suggests, assume Boolean true or false values. We will use variations of Boolean flags to deduce two unifying patterns, one of which unifies "causal" replicated data types, while the second unifies the "last-writer-wins" data types.

[1]Instead of having a single value which depends on the received events, CRDTs are usually defined with a number of *queries* on the CRDT. We choose to forego these for presentation simplicity.

3.1 Causal flags

Take a close look at the following flag definitions:

$$\begin{aligned}\textsc{EWFlag}(E) &= \exists e \in E \cdot e.\mathtt{o} = \mathtt{enable} \\ &\quad \wedge \nexists e' \in E \cdot e'.\mathtt{o} = \mathtt{disable} \wedge e \prec e' \\ \textsc{DWFlag}(E) &= \exists e \in E \cdot e.\mathtt{o} = \mathtt{enable} \\ &\quad \wedge \forall e' \in E \cdot e'.\mathtt{o} = \mathtt{disable} \implies e' \prec e\end{aligned}$$

These are the enable-wins and disable-wins flags, which arbitrate a winner (either `enable` or `disable`) among the *concurrent* operations. More precisely, they pick a winner among events which are not known to be in the causal past of another event. The similarities between these two flags can be made more obvious by rewriting them using the set of maximal events under the causality relation, produced by a function $\max^{\prec}$:

$$\max^{\prec}(E) = \{\, e \in E \mid \nexists e' \in E \cdot e \prec e' \,\}$$

$$\begin{aligned}\textsc{EWFlag}(E) &= \exists e \in \max^{\prec}(E) \cdot e.\mathtt{o} = \mathtt{enable} \\ \textsc{DWFlag}(E) &= \exists e \in \max^{\prec}(E) \cdot e.\mathtt{o} = \mathtt{enable} \\ &\quad \wedge \nexists e' \in \max^{\prec}(E) \cdot e'.\mathtt{o} = \mathtt{disable}\end{aligned}$$

Next we introduce two flag types, even simpler than the EWFlag and DWFlag, that allow only a single (de)activation: the enable-once and disable-once flags (EOFlag and DOFlag). Notice how their definitions are very similar to the rewritten versions of the previous flags, using E instead of $\max^{\prec}(E)$:

$$\begin{aligned}\textsc{EOFlag}(E) &= \exists e \in E \cdot e.\mathtt{o} = \mathtt{enable} \\ \textsc{DOFlag}(E) &= \exists e \in E \cdot e.\mathtt{o} = \mathtt{enable} \\ &\quad \wedge \nexists e' \in E \cdot e'.\mathtt{o} = \mathtt{disable}\end{aligned}$$

3.2 Last-writer-wins flags

We now consider last-writer-wins flags, which rely on a total order ($<$) on process-local wall-clock timestamps. These flags only consider the set of known events with the highest timestamp. If multiple such events exist, they once again need to pick a winner among any conflicting operations. As an example, take a look at the definition for the enable-wins last-writer-wins flag:

$$\begin{aligned}\textsc{LWWEWFlag}(E) &= \exists e \in E \cdot e.\mathtt{o} = \mathtt{enable} \\ &\quad \wedge \nexists e' \in E \cdot e'.\mathtt{o} = \mathtt{disable} \wedge e < e'\end{aligned}$$

As we did for the EWFlag, we can rewrite it to reveal the pattern in common with the EOFlag above, using the set of maximal events, compared by the timestamp total order ($<$):

$$\max^{<}(E) = \{\, e \in E \mid \nexists e' \in E \cdot e < e' \,\}$$

$$\textsc{LWWEWFlag}(E) = \exists e \in \max^{<}(E) \cdot e.\mathtt{o} = \mathtt{enable}$$

The same can be done for defining a LWWDWFlag.

3.3 Partial orderings on events

By now a shared pattern between causal CRDTs and last-writer-wins CRDTs is emerging: both reduce the known set of events to a subset, and then pass that subset on to some other denotation to obtain a value. In both cases the subset is the maximal subset based on an ordering according to some specific property of events: the subset relation over the "knows of" property for causal CRDTs, and the total order over the timestamps for last-writer-wins.

We formalized this idea through a higher-order denotation $\max_{\lessdot,*}$ parametrised over a binary relation $\lessdot$ and a property $*$ of events (typically $\mathtt{t}$ or $\mathtt{k}$), which takes a denotation C and produces a denotation from E to the result of applying C to the maximal subset of events in E:

$$\max_{\lessdot,*}(C) = E \mapsto C(\{\, e \in E \mid \nexists e' \in E \cdot e.* \lessdot e'.* \,\})$$

3.4 Denoting LWW and Causal CRDTs

We can now precisely define the two previously mentioned techniques. The Lww higher-order denotation supposes that every event has a property $\mathtt{t}$ which is the timestamp attached to the event, and which can be compared to the timestamp of another event. The Causal higher-order denotation assumes a property $\mathtt{k}$ for the *knows of* set and the $\subset$ relation, which partially orders pairs of events based on whether or not they are in the causal past of the other.

$$\mathsf{Lww}(C) = \max_{<,\mathtt{t}}(C)$$
$$\mathsf{Causal}(C) = \max_{\subset,\mathtt{k}}(C)$$

We can now simply define the EWFlag to be Causal(EOFlag) and the LWWEWFlag to be Lww(EOFlag). The disable-wins variants are similarly obtained from the DOFlag.

4 HOW SETS RELATE TO FLAGS

As shown in our introductory example, set data types have $\langle \mathtt{add}, a\rangle$ and $\langle \mathtt{rmv}, a\rangle$ operations that take a single element value which is to be added or removed from the set respectively. The difficulty of set CRDTs is that they need to decide what happens when add and rmv operations apply to the same element and possibly occur concurrently—a problem to which various solutions have been proposed. Their definitions in Figure 1 are as follows:

- the grow-only set (GSet) simply ignores removes,
- the two-phase set (2PSet) permanently removes elements if a remove operation for this element is known,
- while the add-wins and remove-wins sets (AWSet and RWSet respectively) mediate between concurrent add and rmv operations by letting either the add or the rmv "win."

All of these set types only differ in the arbitration mechanism. We want to suggest the notion that a set CRDT can be seen as votes on the presence or absence of a value in the set, where presence or absence of a value is decided by grouping the events pertaining to a particular value. This per-element group of events then decides whether an element is in or out.

$$\begin{aligned}
\mathrm{GSet}(E) &= \{\, a \mid e \in E \wedge e.\mathtt{o} = \langle \mathtt{add}, a\rangle \,\} \\
\mathrm{2PSet}(E) &= \{\, a \mid e \in E \wedge e.\mathtt{o} = \langle \mathtt{add}, a\rangle \\
&\qquad \wedge \nexists e' \in E \cdot e'.\mathtt{o} = \langle \mathtt{rmv}, a\rangle \,\} \\
\mathrm{AWSet}(E) &= \{\, a \mid e \in E \wedge e.\mathtt{o} = \langle \mathtt{add}, a\rangle \\
&\qquad \wedge \nexists e' \in E \cdot e'.\mathtt{o} = \langle \mathtt{rmv}, a\rangle \wedge e \prec e' \,\} \\
\mathrm{RWSet}(E) &= \{\, a \mid e \in E \wedge e.\mathtt{o} = \langle \mathtt{add}, a\rangle \\
&\qquad \wedge \forall e' \in E \cdot e'.\mathtt{o} = \langle \mathtt{rmv}, a\rangle \implies e' \prec e \,\}
\end{aligned}$$

Figure 1: Denotations for set CRDTs.

We formalize this notion in the Set(C) function. The function takes a flag denotation C and produces a set denotation which used C to determine whether a value is in the set. To use C, for each a, all $\langle \mathtt{add}, a\rangle$ and $\langle \mathtt{rmv}, a\rangle$ operations of the set data type are transformed to sets of enable and disable flag operations, while preserving the event metadata:

$$\begin{aligned}
\mathsf{Set}(C) &= E \mapsto \{\, a \in \mathsf{elems}(E) \mid C(\mathsf{votes}(E, a)) \,\} \\
\mathsf{elems}(E) &= \{\, a \mid e \in E \wedge (e.\mathtt{o} = \langle \mathtt{add}, a\rangle \vee e.\mathtt{o} = \langle \mathtt{rmv}, a\rangle) \,\} \\
\mathsf{votes}(E, a) &= \{\, e\{\mathtt{o} = \mathtt{enable}\} \mid e \in E \wedge e.\mathtt{o} = \langle \mathtt{add}, a\rangle \,\} \\
&\qquad \cup \{\, e\{\mathtt{o} = \mathtt{disable}\} \mid e \in E \wedge e.\mathtt{o} = \langle \mathtt{rmv}, a\rangle \,\}
\end{aligned}$$

4.1 Maps

As it happens, the concept of a CRDT which groups operations per value already exists in the literature. For example, the observed-removes map from [4] is a map data type which maps keys to instances of a value CRDT. The result is a mapping of all observed keys to values obtained by grouping the events per key and passing these subsets of events to the value denotation.

A map CRDT groups operations op for a specific CRDT type C under each key k through an $\langle \mathtt{apply}, k, op\rangle$ operation, and its value is a partial mapping from keys to values of C. We define a map data type as taking a denotation C to which, for each key k, we pass the result of grouping all events with operation $\langle \mathtt{apply}, k, op\rangle$ while unwrapping operation op:

$$\begin{aligned}
\mathsf{Map}(C) &= E \mapsto \{\, k \mapsto C(\mathsf{ops}(k, E)) \mid k \in \mathsf{keys}(E) \,\} \\
\mathsf{keys}(E) &= \{\, k \mid e \in E \wedge e.\mathtt{o} = \langle \mathtt{apply}, k, op\rangle \,\} \\
\mathsf{ops}(k, E) &= \{\, e\{\mathtt{o} = op\} \mid e \in E \wedge e.\mathtt{o} = \langle \mathtt{apply}, k, op\rangle \,\}
\end{aligned}$$

4.2 Transforming operations and values

If we were to pass a flag to the Map denotation we would have an "almost-set". Instead of $\langle \mathsf{add}, a\rangle$ and $\langle \mathsf{rmv}, a\rangle$ operations we have $\langle \mathsf{apply}, a, \mathsf{enable}\rangle$ and $\langle \mathsf{apply}, a, \mathsf{disable}\rangle$, and instead of a set value we have a mapping.

To solve this problem we can simply transform the operations and values of an existing denotation to make another. To change the value of a CRDT denotation C using a function f from C-values to new values we can use straightforward function composition, i.e. $f \circ C$. Adapting the operations of a denotation C is more complicated: we need to intercept the set of events and update the o property. For this we define function $\mathsf{map}_o(f, C)$ which takes an operator-transforming function f and a denotation C, and translates operations of input events to those of the existing denotation C, while keeping all remaining properties of events unmodified:

$$\mathsf{map}_o(f, C) = E \mapsto C(\{\, e\{\mathsf{o} = f(e.\mathsf{o})\} \mid e \in E \,\})$$

4.3 Sets based on flags and maps

Now that we have a way to associate CRDTs to values, defining the behavior of set CRDTs is a trivial matter of translating the $\langle \mathsf{add}, a\rangle$ and $\langle \mathsf{rmv}, a\rangle$ operations to $\langle \mathsf{apply}, a, \mathsf{enable}\rangle$ and $\langle \mathsf{apply}, a, \mathsf{disable}\rangle$ operations for the particular flag denotation which decides whether a is present or absent in the set. Through function composition and map_o we can now redefine the Set(C) higher-order denotation as taking a flag denotation C and using it as the value CRDT for a Map higher-order denotation, translating operations and values appropriately:

$$\begin{aligned}
\mathsf{Set}(C) &= f_v \circ \mathsf{map}_o(f_o, \mathsf{Map}(C)) \\
f_o(\langle \mathsf{add}, a\rangle) &= \langle \mathsf{apply}, a, \mathsf{enable}\rangle \\
f_o(\langle \mathsf{rmv}, a\rangle) &= \langle \mathsf{apply}, a, \mathsf{disable}\rangle \\
f_v(v) &= \{\, a \mid (a \mapsto \mathsf{True}) \in v \,\}
\end{aligned}$$

5 EXPRESSIVE POWER

To convince the reader of the expressive power of our Set, Lww, and Causal higher-order denotations we now define a large number of CRDTs using them. However, we first define a few more primitive data types, namely a counter CRDT (the PNCounter) and the last-writer-wins register (LWWRegister). This last data type is used to define the last-writer-wins flag (LWWFlag). Using the LWWFlag, EOFlag, DOFlag, Set, Lww, and Causal denotations we then define a complete range of flag and set data types. Finally we introduce the multi-value register, and show that it is in effect no different from a grow-only set with Causal semantics.

5.1 PNFlag and PNCounter

An early solution for a set to which the same elements could be added and removed multiple times was the positive-negative set [1]. Since we know how to define sets from any flag, we will first define the positive-negative flag, or PNFlag.

This flag is enabled if there are more known enable operations than $\mathsf{disable}$ operations. The counting aspect is of course very reminiscent of the well-known positive-negative counter CRDT (PNCounter) which we will define first so that we can reuse it.

The PNCounter counts inc and dec operations and is integer-valued, where the value is the difference in number between known increments and decrements of the counter:

$$\begin{aligned}
\text{PNCounter}(E) = {} & |\{\, e \in E \mid e.\mathsf{o} = \mathsf{inc} \,\}| \\
& - |\{\, e \in E \mid e.\mathsf{o} = \mathsf{dec} \,\}|
\end{aligned}$$

Defining the PNFlag in terms of the PNCounter is easy: all that needs to happen is for enable and $\mathsf{disable}$ operations to be translated to inc and dec operations, and for the value of the PNCounter to be checked if it is positive:

$$\begin{aligned}
\text{PNFlag} &= (n \mapsto n > 0) \circ \mathsf{map}_o(f, \text{PNCounter}) \\
f(o) &= \begin{cases} \mathsf{inc}, & \text{if } o = \mathsf{enable} \\ \mathsf{dec}, & \text{if } o = \mathsf{disable} \end{cases}
\end{aligned}$$

5.2 LWWRegister and LWWFlag

The last-writer-wins register is one of the first attempts at implementing an eventually-consistent mutable value. It attaches a wall-clock timestamp to events to impose a total order on them. This can be achieved by using combining timestamps with a globally unique site-identifier for each process, where the identifier arbitrates in case of identical wall-clock timestamps. Register data types have $\langle \mathsf{write}, a\rangle$ operations to set the register. The LWWRegister simply picks the a with the highest timestamp, or has a $\bot$ value if no $\langle \mathsf{write}, a\rangle$ operations are known. Assuming events have a property T which totally orders events, we can define the data type by using our max higher-order denotation defined earlier:

$$\text{LWWRegister}(E) = \begin{cases} a, & \text{if } \mathsf{max}_{<,\mathsf{T}}(E) = \{\, \langle \mathsf{write}, a\rangle \,\} \\ \bot, & \text{otherwise} \end{cases}$$

The last-writer-wins flag is simply a Boolean-valued register, which we can define by transforming the values of a LWWRegister:

$$\begin{aligned}
\text{LWWFlag} &= f_v \circ \mathsf{map}_o(f_o, \text{LWWRegister}) \\
f_o(a) &= \langle \mathsf{write}, a\rangle \\
f_v(a) &= (a = \mathsf{enable})
\end{aligned}$$

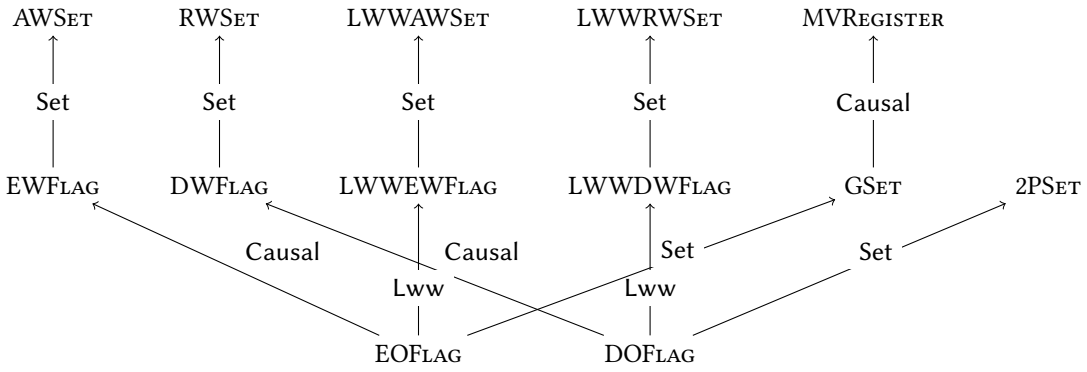

Figure 2: Hierarchy of data types that derive from the two basic flags: EOFlag and DOFlag.

5.3 Causal and LWW flags

Below is the table of flags based on EOFlag and DOFlag, wherein variations are obtained using Lww and Causal. This list completes our catalog of flags which we use to define our catalog of sets in the next section:

C	Causal(C)	Lww(C)
EOFlag	EWFlag	LWWEWFlag
DOFlag	DWFlag	LWWDWFlag

5.4 A full set of sets

Below is a table showing how all set data types found in the literature can now be expressed based on the previously defined flag data types and the Set higher-order denotation:

C	Set(C)
EOFlag	GSet
DOFlag	2PSet
PNFlag	PNSet
LWWFlag	LWWSet
LWWEWFlag	LWWAWSet
LWWDWFlag	LWWRWSet
EWFlag	AWSet
DWFlag	RWSet

Figure 2 presents the combined effect of all these transformations.

5.5 Multi-value register

A register data type which improves upon the LWWRegister by taking into account causality rather than relying on totally ordered timestamps is the multi-value register or MVRegister. Rather than having a single value, its value is the set of values corresponding to the concurrent $\langle \texttt{write}, a \rangle$ operations. To define an MVRegister, we can reuse the GSet, since it is already set-valued. Making the GSet a causal CRDT ensures that only the concurrent $\langle \mathsf{add}, a \rangle$ operations are part of the value of the set:

$$\text{MVRegister} = \mathsf{map}_o(\langle \texttt{write}, a \rangle \mapsto \langle \mathsf{add}, a \rangle, \text{Causal}(\text{GSet}))$$

5.6 The Redis register

Finally, the Redis register is a causal last-writer-wins register, where the last-writer-wins behavior is only used to disambiguate in case multiple concurrent operations exist. Defining it is as simple as passing the LWWRegister to the Causal higher-order denotation:

$$\text{RedisRegister} = \text{Causal}(\text{LWWRegister})$$

6 RELATED WORK

We are not the first to suggest that more complex CRDTs can be composed out of simpler ones [3], nor are we the first to formalize the behavior of CRDTs [3, 6, 7]. The work on composite data types by Gotsman and Yang [3] formalized the use of CRDT-valued key-value stores with causal transaction support to build new CRDTs. It assumes a key-value store has support for various primitive data types such as the LWWSet, and models a programming language with causal transactions to compose these primitives into more complex CRDTs, such as a social graph data type. What distinguishes our work is that we found structure in CRDTs which are normally considered primitive, thereby reducing the number of primitive CRDTs (for this paper) to just the EOFlag, DOFlag, PNCounter, and LWWRegister.

Our denotational framework captures the same generality of the arbitration and visibility relations on events of [7] but we forego the possibility of an operation on a CRDT instance returning a value, and the possibily of multiple queries on a data type. Instead of the latter we opted for a single value derived from the set of known events at a replica. Both of these modifications result in a slightly less general denotation but one which we consider more naturally composable. Treating the arbitration and visibility relations as embedded in the set

of events instead of as explicit arguments to the denotation as in [7] results in a notation which is more succinct for our purposes as well.

Gaducci et al. [6] also take a denotational look at CRDT semantics, basing their analysis on causal graphs of events, similarly to how our events form a causal graph through the "knows of" property. Interestingly, the authors take a view of CRDTs in which definitions for abstract data types such as the set and register are refined using a conflict resolution policy to obtain concrete data types. This is akin to our Set higher-order denotation which takes a flag to decide on conflicts. However, unlike the flags passed to our Set denotation, their conflict resolution mechanisms are not CRDTs themselves.

Previous work on formalizing CRDTs has focused on proving the correctness of a CRDT implementation based on formalized descriptions of CRDT semantics [7]. We on the other hand do not consider implementation at all, but we hope to have made future correctness efforts easier through our work: an implementation could be based on semantic patterns such as Lww and Set, and use data structures such as Map to implement CRDTs, thereby following the same structure as the formal definition.

Finally, we are not the first to attempt to catalog CRDTs either [1], but as far as we know we are the first to do so based on semantic patterns.

7 FUTURE WORK

This paper is a result of attempting to implement CRDTs in multiple styles (state-based and operation-based) and noticing similar patterns emerge. In future work we plan to describe how these patterns lend themselves well to compilation if treated as a definition language for eventually consistent data types. Along with the definition for such a language we plan to publish how it can be compiled to efficient implementations in any current CRDT implementation strategy, be it operation-based or state-based CRDTs.

ACKNOWLEDGMENTS

This work was partially supported by the European Union H2020 LightKone project under grant 732505 (https://www.lightkone.eu/). Adriaan Leijnse would like to acknowledge Protocol Labs, Inc. for their graciously provided funding.

REFERENCES

[1] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011.

[2] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based crdts operation-based. In Kostas Magoutis and Peter Pietzuch, editors, *Distributed Applications and Interoperable Systems*, pages 126–140, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[3] Alexey Gotsman and Hongseok Yang. Composite replicated data types. In Jan Vitek, editor, *Programming Languages and Systems*, pages 585–609, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[4] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *Journal of Parallel and Distributed Computing*, 111:162 – 173, 2018.

[5] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] Fabio Gadducci, Hernán Melgratti, and Christian Roldán. A denotational view of replicated data types. In Jean-Marie Jacquet and Mieke Massink, editors, *Coordination Models and Languages*, pages 138–156, Cham, 2017. Springer International Publishing.

[7] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. *SIGPLAN Not.*, 49(1):271–284, January 2014.